Tournaments for mutual exclusion

Hesselink, Wim H.

Published in:
Formal Aspects of Computing

DOI:
10.1007/s00165-016-0407-x

Publication date:
2017

DOI 10.1007/s00165-016-0407-x

Formal Aspects of Computing (2017) 29: 833-852

Formal Aspects of Computing

Tournaments for mutual exclusion: verification and concurrent complexity

Wim H. Hesselink

Johann Bernoulli Institute for Mathematics and Computer Science University of Groningen, P.O.Box 407, 9700 AK Groningen, The Netherlands

Abstract. Given a mutual exclusion algorithm MXd for $d \geq 2$ threads, a mutual exclusion algorithm for $N > d$ threads can be built in a tree of degree d with N leaves, with the critical section at the root of the tree. This tournament solution seems obviously correct and efficient. The present note proves the correctness, and formalizes the efficiency in terms of concurrent complexity by means of Bounded Unity. If the tree is balanced, the throughput is logarithmic in N. If moreover MXd satisfies FCFS (first-come first-served), the worst case individual delay of the tournament algorithm is of order N. This is optimal.

Keywords: Mutual exclusion, UNITY, Concurrency, Shared variables, Concurrent complexity

1. Introduction

The problem of mutual exclusion for N threads or processes was proposed by Dijkstra [Dij65] in 1965. His solution was improved by Knuth [Knu66] and De Bruijn [dB67]. In 1974, Lamport [Lam74] proposed his Bakery Algorithm. In all these solutions, the competing threads inspected the intentions of the other threads by reading an array of size N. In 1977, Peterson and Fischer [PF77] proposed the first tournament solution, in which a competing thread need not inspect the intentions of all other threads.

In general, a tournament solution of mutual exclusion for N threads is based on an auxiliary solution MXd for $d \geq 2$ threads, where d is a small number, usually $d = 2$. One constructs a tree with N leaves in which every internal node has at most d children. Initially, the threads are located in the leaves of the tree. In order to reach the critical section, a thread repeatedly performs the entry protocol of MXd in competition with the threads located in the siblings of its current node. Each time it completes an entry protocol, the thread moves to the parent of its current node. When it reaches the root of the tree, it has access to the critical section. After this, the thread moves back along its root path to execute the exit protocols of MXd.

Recent experiments of Buhr et al. [BDH15, BDH16] show that tournament algorithms for mutual exclusion can perform almost as good as the hardware assisted MCS algorithm of Mellor-Crummey and Scott [MCS91]. The challenge for the present paper is to complement this experimental result with a theoretical estimate of the concurrent complexity of tournament algorithms, and to see how this depends on the choice of MXd.

Correspondence and offprint requests to: W. H. Hesselink, E-mail: w.h.hesselink@rug.nl

Indeed, one of the issues is the choice of MXd. If one takes $d = 2$, the main candidates are Dekker's algorithm [BDH16] or Peterson's algorithm [Pet81]. Zhang et al. [ZYC96] propose and test tournament solutions with $d > 2$, using the Burns-Lamport solution [Bur81, Lam86] for d threads. Kessels [Kes82] has nicely described how to treat the shared variables and the private variables of the subsequent incarnations of MXd, and how to relate the identity of the thread with the identity of its current node, and the siblings and parent of the current node.

In this article, we prove the correctness and determine the concurrent complexity of a tournament solution, TMX, based on an abstract auxiliary algorithm, MXd, for d threads. The idea of the tournament solution to mutual exclusion is fairly simple. Yet, the threads act concurrently and can be all in different stages of the tournament. In this respect, the situation differs completely from orderly tournament in tennis or chess, where the competitors proceed more or less in lockstep. Another difficulty of the proof of mutual exclusion is to relate the N threads that participate in the tournament to the d branches that can participate in the local algorithm MXd, and to transfer the mutual exclusion of MXd to the tournament.

Concurrent complexity is a complexity measure for concurrent algorithms based on the time unit *round*. Roughly speaking, a round is an execution fragment in which every thread is scheduled at least once, and is executed or found to be disabled. See Sect. 4.1 for details.

With respect to concurrent complexity, we distinguish throughput and individual progress. Throughput is the total number *nc* of critical sections executed, compared to the total number *nr* of rounds. It is proved that, if the tree is balanced, the throughput factor nr/nc has an upper bound that is logarithmic in N for long executions in which always at least one thread is competing. Individual progress counts the maximal number of rounds for a competing thread to become idle again. If the tree is balanced and MXd has the FCFS property (first-come first-served), it is proved that the number of rounds a competing thread may need to become idle again is bounded by an expression linear in N.

We make the usual assumption that the scheduling is weakly fair: if some thread is continuously enabled from some time onward, it will eventually be scheduled and do a step. With respect to the grain of atomicity, we keep to the *principle of single critical reference*, e.g. [OG76, (3.1)], [AdBO09, p. 273]): in every atomic step at most one shared variable is inspected or modified (not both). Actions on private variables and on history variables can be added to atomic commands because they never lead to interference. In this article, the principle is hardly relevant, however, because almost all shared variables are history variables.

A conceptual difficulty of the article is that it is based on an arbitrary mutual exclusion algorithm MXd. It therefore needs a faithful model of MXd that offers precisely what MXd can be assumed to offer. This is done by modelling MXd with auxiliary variables, which do not occur in the implementations of MXd, but which can be added as history variables to any implemation. As they are history variables, actions on them can be combined in coarse grain atomic commands if this is justified by the specification of MXd. With respect to safety, the challenge is to prove that the tournament algorithm guarantees mutual exclusion for N threads based on the assumption that MXd guarantees mutual exclusion for d children. With respect to progress, the challenge is to estimate progress of the tournament in terms of estimates of progress for MXd.

We have verified the technical assertions of the article with the proof assistant PVS [OSRSC01]. Readers familiar with PVS may wish to inspect the proof scripts at our web page [Hes16b], but we do not expect the general reader to do so. The starting point of this verification is the tree described in Sect. 2.1 and the transition system of Fig. 1. From Sect. 3.1 onward, everything, except for Sect. 6.1, has been verified with PVS.

Overview

Section 2 discusses the programming of the tournament. In Sect. 3, a transition system is developed to analyse and discuss the algorithm. This section also treats the safety properties of the algorithm: the proof of mutual exclusion, and some other invariants. Section 4 prepares the proofs of progress. In particular, it presents Bounded Unity, a formal system to prove and quantify progress properties. It also introduces numerical parameters for the auxiliary algorithm MXd, and determines these for the case that MXd is Peterson's algorithm. In Sect. 5, throughput is determined as announced above. Section 6 treats individual progress. Conclusions are drawn is Sect. 7.

2. Programming the tournament

Let *MXd* be a mutual exclusion algorithm for $d \geq 2$ threads. *MXd* is used to form a mutual exclusion algorithm for $N > d$ threads by means of a d-ary tree with N leaves, one leaf per thread. When a thread is idle, it resides in its leaf. When it needs to enter the critical section, it goes from its leaf along the root path to the root. At each internal node, *MXd* is used to determine which of the competing threads can proceed. A thread at the root can enter the critical section. Afterwards, the thread traverses its root path back to its leaf and releases the nodes along the path by the exit protocol of *MXd*.

2.1. The tournament

The design of the tournament starts with the design of the d-ary tree. There are many ways to represent trees. The set $Child = \{i \in \mathbb{N} \mid i < d\}$ is used here to number the children of the nodes. The tree is given by a set *Node* with a specific element $root \in Node$, and three functions

$$\begin{aligned} &\delta : Node \to \mathbb{N} , \\ &path : Node \times \mathbb{N} \to Node , \\ &sib : Node \times \mathbb{N} \to Child . \end{aligned}$$

Function δ gives the depth of the node, i.e., the distance to the root. Function *path* gives the root path of the node starting from *root*. We postulate that, for all nodes n and natural numbers j,

$$\begin{aligned} &path(n, 0) = root , \\ &path(n, \delta(n)) = n , \\ &\delta(path(n, j)) = \min(j, \delta(n)) . \end{aligned}$$

It follows that function *path* satisfies the simple rule:

$$path(n, j) = path(n', k) \ \wedge \ j \leq \delta(n) \ \wedge \ k \leq \delta(n') \ \Rightarrow \ j = k .$$

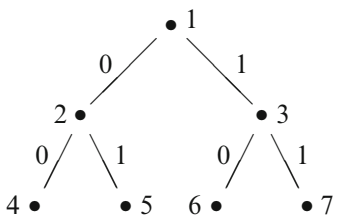
For $j < \delta(n)$, the value $sib(n, j) \in Child$ is the number of node $path(n, j + 1)$ as a child of node $path(n, j)$. The sibling numbers characterize the children of a node in the sense that, for all natural numbers j, and nodes n and n':

$$\begin{aligned} (1) \qquad & j < \delta(n) \ \wedge \ j < \delta(n') \ \wedge \ path(n, j) = path(n', j) \ \wedge \ sib(n, j) = sib(n', j) \\ & \Rightarrow \quad path(n, j + 1) = path(n', j + 1) . \end{aligned}$$

The leaves of the tree are the nodes not in the root path of any other node:

$$(2) \qquad p \in Leaf \quad \equiv \quad (\forall \ j \in \mathbb{N}, n \in Node : j \leq \delta(n) \ \wedge \ path(n, j) = p \ \Rightarrow \ n = p) .$$

Example. The diagram shows a binary tournament tree for 4 threads.
The nodes are numbered from 1, the root, up to 7.
The six edges are labeled with sibling numbers 0 and 1.
The root path of node 6 (a leaf) consists of the nodes
$path(6, 0) = 1$, $path(6, 1) = 3$, $path(6, 2) = 6$,
with sibling numbers $sib(6, 0) = 1$ and $sib(6, 1) = 0$.



Mutual exclusion is the problem that several concurrent threads repeatedly need exclusive access to a shared resource, the *critical section CS*. When thread p needs the *CS* while another thread occupies it, thread p needs to wait. Therefore, every mutual exclusion algorithm has a function *entry* to guard the *CS*, and a function *exit* to make it available again. In the tournament algorithm, we need to distinguish the *entry* and *exit* functions of the nodes of the tree from the *entry* and *exit* functions of the tournament.

Let $IntNode = Node \setminus Leaf$ be the set of the internal nodes of the tree. For the tournament algorithm, the leaves of the tree are used as thread identifiers. In other words, *Leaf* is identified with *Thread*, the set of the threads. The algorithm uses *MXd* at each internal node to choose between competing siblings. It is therefore assumed that algorithm *MXd* has the entry and exit functions

$Entry(n : IntNode, i : Child)$;
$Exit(n : IntNode, i : Child)$,

where the argument n serves as a pointer to the shared space reserved for the incarnation of the algorithm, and i holds the child identity. Alternatively, we could regard node n as an object with the methods *Entry* and *Exit*.

The entry and exit functions of the tournament both use a private variable ℓ (level) for the depth of the current node. The entry function is given by

$EntryTMX(p : Thread) =$
  var $\ell := \delta(p) - 1$;
  while $\ell \geq 0$ **do**
    $Entry(path(p, \ell), sib(p, \ell))$;
    $\ell := \ell - 1$;
  end

At node $path(p, \ell)$, thread p thus uses its sibling number $sib(p, \ell) \in Child$ as thread identifier.

After the critical section, the thread exits from the nodes it has occupied in the exit protocol:

$ExitTMX(p : Thread) =$
  var $\ell := -1$;
  while $\ell < \delta(p) - 1$ **do**
    $\ell := \ell + 1$;
    $Exit(path(p, \ell), sib(p, \ell))$;
  end

2.2. Examples for the algorithm MXd at the nodes

For concreteness, one can take Peterson's algorithm [Pet81] for MXd. In this case, $d = 2$ and $Child = \{0, 1\}$, and there are shared variables

$\mathtt{turn}[IntNode] : Child := 0$;
$\mathtt{flag}[IntNode, Child] : bool := false$.

The entry function is

$EntryP(n : IntNode, i : Child) =$
  $\mathtt{flag}[n, i] := true$;
  $\mathtt{turn}[n] = 1 - i$;
  await $\neg\, \mathtt{flag}[n, 1 - i] \ \vee\ \mathtt{turn}[n] = i$;
end

The exit function is

$ExitP(n : IntNode, i : Child) =$
  $\mathtt{flag}[n, i] := false$;
end

Alternatively, one can take Dekker's algorithm [BDH16] for MXd. Again $d = 2$. The algorithm uses the same shared variables, and the entry function

$EntryD(n : IntNode, i : Child) =$
  loop
    $\mathtt{flag}[n, i] := true$;
    if $\neg\, \mathtt{flag}[n, 1 - i]$ **then return endif** ;
    if $\mathtt{turn}[n] = i$ **then**
      await $\neg\, \mathtt{flag}[n, 1 - i]$; **return** ;
    endif ;
    $\mathtt{flag}[n, i] := false$;
    await $\mathtt{turn}[n] = i$;
  endloop
end

The exit function for Dekker's algorithm is

$$
\begin{array}{l}
\mathit{ExitD}(n : \mathit{IntNode}, i : \mathit{Child}) = \\
\quad \mathtt{turn}[n] := 1 - i \ ; \\
\quad \mathtt{flag}[n, i] := \mathit{false} \ ; \\
\textbf{end}
\end{array}
$$

Buhr et al. [BDH15, BDH16] describe implementations in C of tournament algorithms based on the algorithms of Peterson and Dekker, respectively, and give performance results.

2.3. Taking a more abstract view

The aim is to prove safety and progress for every tournament algorithm. This requires abstraction from the details of *MXd*, while retaining its mutual exclusion property. We therefore introduce a history variable $\mathtt{mu}[\mathit{IntNode}] : \mathit{Child} \cup \{\bot\}$ with $\mathtt{mu}[n] = \bot$ initially for all internal nodes n. When child i enters the critical section of *MXd* at node n, it sets $\mathtt{mu}[n] := i$. It resets $\mathtt{mu}[n] := \bot$ before exiting. As *MXd* guarantees mutual exclusion, child i does not enter the critical section when $\mathtt{mu}[n] = j$ for some sibling $j \neq i$. The algorithm *MXd* is thus modelled by the commands

$$
\begin{array}{l}
\mathit{Entry}(n : \mathit{IntNode}, i : \mathit{Child}) = \\
\quad \textbf{await } (\mathtt{mu}[n] = \bot \ \vee \ \mathtt{mu}[n] = i) \textbf{ then } \mathtt{mu}[n] := i \ ; \\
\textbf{end}
\end{array}
$$

$$
\begin{array}{l}
\mathit{Exit}(n : \mathit{IntNode}, i : \mathit{Child}) = \\
\quad \mathtt{mu}[n] := \bot \ ; \\
\textbf{end}
\end{array}
$$

This algorithm guarantees mutual exclusion between threads in *Child*, because $\mathtt{mu}[n] = i$ when thread $i \in \mathit{Child}$ has executed $\mathit{Entry}(n, i)$ and not yet $\mathit{Exit}(n, i)$.

Algorithm *MXd* cannot distinguish different threads that approach a node via the same child. Therefore, by the disjunct $\mathtt{mu}[n] = i$, the **await** statement allows such threads to enter concurrently. The correctness proof of the tournament algorithm will show, however, that this does not happen.

In order to see that the two algorithms of Sect. 2.2 implement this abstract mutual exclusion algorithm, one augments them with the history variable mu with initially $\mathtt{mu}[n] = \bot$ for all nodes n, and modifies mu by

$$
\begin{array}{l}
\mathtt{mu}[n] := i \ \text{ at the end of } \mathit{Entry}(n, i) \ ; \\
\mathtt{mu}[n] := \bot \ \text{ at the start of } \mathit{Exit}(n, i) \ .
\end{array}
$$

As the algorithm *MXd* guarantees mutual exclusion, this ensures that $\mathit{Entry}(n, i)$ never modifies $\mathtt{mu}[n]$ unless $\mathtt{mu}[n] = \bot$ holds. Therefore, the body of *Entry* can be regarded as a single atomic command.

3. Mutual exclusion and other safety properties verified

This section verifies that the tournament algorithm satisfies mutual exclusion for N threads, and other safety properties. For this purpose, the entry and exit functions are combined with an abstract application environment and formalized in a transition system in Sect. 3.1. This transition system is also the starting point for our mechanical verification by means of the proof assistant PVS [OSRSC01].

In Sect. 3.2, it is proved that the tournament algorithm satisfies mutual exclusion. Section 3.3 presents some further invariants, which are needed for the proofs of progress.

3.1. The transition system and its first invariants

In order to analyse the algorithm, we form the transition system of Fig. 1, in which the entry and exit functions of Sect. 2.3 are applied by an unknown environment that consists of a noncritical section *NCS* and a critical section *CS*. The program fragment *NCS* need not terminate, but when it terminates it calls the entry function. A thread is said to be *idle* when it is at *NCS*, i.e., at line 11. Otherwise, it is said to be *competing*. The fragment *CS* must be performed under mutual exclusion. It is assumed to terminate.

```
Tournament(p : Thread) =
    var ℓ := δ(p) − 1 ;
11      NCS ;
12      while ℓ ≥ 0 do
15          await (mu[path(p, ℓ)] = ⊥ ∨ mu[path(p, ℓ)] = sib(p, ℓ)) then
                mu[path(p, ℓ)] := sib(p, ℓ) ; ℓ := ℓ − 1
        endwhile ;
16      CS ;
17      while ℓ < δ(p) − 1 do
18          ℓ := ℓ + 1 ; mu[path(p, ℓ)] := ⊥
        endwhile ;
19      cnt := cnt + 1 ; goto 11 .
```

Fig. 1. Transition system of the tournament

The line numbers in Fig. 1 start arbitrarily at 11, for the ease of modification during development. The line numbers 13 and 14 are skipped to leave room for additional commands, as will be used in Sect. 6. In line 19, an auxiliary private variable *cnt* is incremented, for the analysis of progress. The backward jump in line 19 is included to allow repeated application of the algorithm. Every thread, q, has an implicit private variable $pc.q$, its program counter, which ranges between 11 and 19. This program counter is modified implicitly at the end of every command in the obvious way. We write q **at** k to abbreviate $pc.q = k$. Similarly, if S is a set of line numbers, q **in** S means $pc.q \in S$.

The line numbers indicate atomic steps. For instance, line 12 corresponds to the atomic command

$$12 \qquad \textbf{if } \ell \geq 0 \textbf{ then goto } 15 \textbf{ else goto } 16 \textbf{ endif} \ ,$$

while the repetition is enforced by extending command 15 implicitly with the backward jump to line 12, as is done by every compiler.

Line 15 may seem to violate the principle of single critical reference (see Sect. 1): step 15 reads the shared variable $\mathtt{mu}[path(p, \ell)]$ and optionally writes it. This however is allowed because mu is a history variable, introduced to abstract from the details of the implementing algorithm *MXd*.

The lines 15 and 18 also contain assignments to the private variable $\ell.p$. This is allowed because commands about private variables can never lead to interference. It is possible to separate the assignments to $\ell.p$ from the lines 15 and 18, but this would force us to more or more complicated invariants lower down.

The transition system has the shared variables $\mathtt{mu}[n]$ for all internal nodes n, and for each thread q private variables $\ell.q$ and $pc.q$. In the initial state, every internal node n has $\mathtt{mu}[n] = \bot$, and every thread q has $pc.q = 11$ and $\ell.q = \delta(q) - 1$.

As a preparation of the proof of mutual exclusion, we note the following invariants concerning the value of ℓ of thread q:

Iq0: $\quad -1 \leq \ell.q \leq \delta(q) - 1$,
Iq1: $\quad q$ **in** $\{13 \ldots 15\} \ \Rightarrow \ 0 \leq \ell.q$,
Iq2: $\quad q$ **at** $16 \ \Rightarrow \ \ell.q = -1$,
Iq3: $\quad q$ **at** $18 \ \Rightarrow \ \ell.q < \delta(q) - 1$,
Iq4: $\quad q$ **in** $\{11, 19\} \ \Rightarrow \ \ell.q = \delta(q) - 1$.

When postulating such invariants, we implicitly mean the universal quantification over all free variables (here q only). Predicate *Iq0* is invariant: it holds initially because of $\ell.q = \delta(q) - 1$, and it is preserved in every step. Indeed, it is threatened only by the steps 15 and 18 that modify ℓ. It is preserved by these steps because of *Iq1* and *Iq3*, respectively. The predicates *Iq1* and *Iq3* are inductive: preservation can be proved without using other invariants in the precondition. Predicate *Iq2* is threatened by the jump from line 12 to line 16 when $\ell.p < 0$. It is preserved because of *Iq0*. Similarly, predicate *Iq4* is threatened only by the jump from line 17 to 19, and is preserved because of *Iq0*.

3.2. Proving mutual exclusion

The main proof obligation is mutual exclusion, i.e., that two different threads q and r are never concurrently at the critical section *CS*. As *CS* is line 16, this property is expressed by the invariant

MX: $\quad q$ **at** $16 \ \wedge \ r$ **at** $16 \ \Rightarrow \ q = r$.

How to prove this invariant? The informal argument is as follows. The algorithm guarantees that every node is occupied by at most one thread, and this implies mutual exclusion because thread p only enters the critical section *CS* when it has $\ell.p = -1$ or equivalently when it occupies the root of the tree.

To formalize this argument, thread p is defined to *occupy* the nodes $path(p, j)$ with $\ell.p < j \leq \delta(p)$. Therefore, the property that every node is occupied by at most one thread, is expressed by

MX1: $\quad \ell.q < j \leq \delta(q) \ \wedge \ \ell.r < j \leq \delta(r) \ \wedge \ path(q, j) = path(r, j) \ \Rightarrow \ q = r$.

This predicate implies *MX* because when the threads q and r are both at line 16, *Iq2* implies that $\ell.q = \ell.r = -1$. We can therefore use $j = 0$ in *MX1* and use that $path(q, 0) = root = path(r, 0)$.

The validity of *MX1* depends on the properties of the variables `mu`. Indeed, in order to prove predicate *MX1*, we postulate the invariant

Iq5: $\quad \ell.q < j < \delta(q) \ \Rightarrow \ \texttt{mu}[path(q, j)] = sib(q, j)$.

Predicate *MX1* follows from *Iq5*. This is proved as follows. Let q, r, j be such that the antecedent of *MX1* holds while $q \neq r$. If $j < \delta(q)$ and $j < \delta(r)$, the invariant *Iq5* for both q and r implies that $sib(q, j) = sib(r, j)$, and hence that $path(q, j + 1) = path(r, j + 1)$ by Formula (1). We can therefore replace j by $j + 1$. Repeating this, we reach $j = \delta(q)$ or $j = \delta(r)$. As q and r are leaves, Formula (2) then implies that $q = r$.

Predicate *Iq5* holds initially because of $\ell.q = \delta(q) - 1$. It is threatened only by step 18 of a thread $p \neq q$. It is preserved by this step because of *MX1* with $r := p$ and $j := \ell.p + 1$, and *Iq0* and *Iq3*. Note that this is not a case of circular reasoning. Indeed, we may assume that all postulated invariants hold in the precondition of the step. Therefore, *MX1* holds in the precondition of the step. This proves that the step at line 18 preserves *Iq5*. This concludes the proof of the invariant *Iq5*, and of the derived invariants *MX1* and *MX*, and therefore of mutual exclusion.

Note that the easy invariants *Iq0*, ..., *Iq4* were presented bottom-up, while the more complicated invariants *MX*, *MX1*, *Iq5* were derived in a top-down fashion.

3.3. Invariants for progress

As a preparation of the proofs of progress in the Sects. 5 and 6, we introduce two more invariants.

Firstly, we have the obvious invariant that every thread is always at one of the program locations:

Kq0: $\quad q$ **in** $\{11, 12\} \cup \{15 \ldots 19\}$.

We also need an inverse of the implication of *Iq5*: whenever some node n has $\texttt{mu}[n] \neq \bot$, there is a thread r that occupies the node. This is expressed in

Kq1: $\quad \texttt{mu}[n] \neq \bot \ \Rightarrow \ \exists \ r : \ell.r < \delta(n) < \delta(r) \ \wedge \ n = path(r, \delta(n))$.

This predicate holds initially because of $\texttt{mu}[n] = \bot$. When thread p executes line 15, it becomes a witness of the existential quantification because of *Iq0* and *Iq1*. In the other steps the witness need not be changed.

It is easy to understand why the invariant *Kq1* is needed for the proof of progress: if, at some time, $\texttt{mu}[n] \neq \bot$ would hold without an occupying thread, this would remain true and all idle threads with node n in the root path would never be able to pass node n and reach the critical section.

4. Preparation of the treatment of progress

Roughly speaking, progress of an algorithm means that the aims of the algorithm will be established when the algorithm has done sufficiently many steps. This can be expressed and proved in various forms of temporal logic. Experience has taught that admitting operational arguments for a specific concurrent algorithm leads to

confusing and unreliable proofs. It is therefore better to use a theory that treats the operational arguments at a general level, and allows assertional reasoning for the specific algorithm.

Progress for the tournament algorithm is proved here with Bounded Unity [Hes99, Hes15]. This is a theory to prove and estimate progress by assertional means. It is a quantitative version of UNITY [CM88, Mis01]. To distinguish the two versions, the acronym UNITY is used only for the original version.

Section 4.1 gives the operational background. It is used here only to interpret the questions and results. In the mechanical proofs, it is used as a foundation. Section 4.2 presents the basic proof rules of Bounded Unity. It also gives a result about the growth of state functions that is used later.

In Sect. 4.3, progress of the single commands of Fig. 1 is discussed. In particular, parameters are introduced for the concurrent complexity of the abstract commands. Section 4.4 determines these parameters for the case that *MXd* is Peterson's algorithm.

4.1. The operational foundation

The state of the system is given by the values of all shared and private variables. Let X be the set of all states. If P is a predicate on the state, P is also regarded as a subset of X, viz. as the set of the states that satisfy P. In particular, the invariants of Sect. 3 are subsets of X. Let INV be the intersection of all invariants obtained. Although predicates on X are now identified with subsets of X, we mostly keep to predicate notation: $P \wedge Q$ and $P \vee Q$ for conjunction and disjunction, $\neg P$ for negation. The main exception is that, if P and Q are predicates, we use set inclusion $P \subseteq Q$ to express that P implies Q.

Let *step* be the binary relation on X such that $(x, y) \in \mathit{step}$ iff state x has some transition to state y or $x = y$. An *execution fragment* of length $n \geq 0$ is a nonempty finite sequence $(x_0, \ldots, x_n)$ in the set INV such that $(x_i, x_{i+1}) \in \mathit{step}$ for all $0 \leq i < n$. Two execution fragments can be concatenated iff the final state of the first fragment equals the initial state of the second fragment.

In Bounded Unity, it is assumed that there is a set of threads, and that every thread p has a binary relation $\mathit{fwd}(p) \subseteq \mathit{step}$. The steps in $\mathit{fwd}(p)$ are called the *forward* steps of p. Thread p is said to be *enabled* when it can do a forward step. This is formalized by defining

$$\mathit{ena}(p) = \{x \mid \exists\ y : (x, y) \in \mathit{fwd}(p)\}\ .$$

For example, a thread at a statement **await**(B) is not enabled when B is false.

By definition, relation *step* is reflexive. The relations $\mathit{fwd}(p)$ are usually not reflexive. In our application, Fig. 1, we define the forward steps to be those that start in the lines 12, 17, and 19. The steps at 15 and 18 are abstract steps, presumably implemented using forward steps. The steps at 11 and 16 are application dependent abstract steps.

Thread p is defined to *occur in* an execution fragment $(x_0, \ldots x_n)$ iff there is an index i with $0 \leq i < n$, and $(x_i, x_{i+1}) \in \mathit{fwd}(p)$ or $x_i \notin \mathit{ena}(p)$. An execution fragment is called a *round* iff it contains an occurrence for every thread. Informally speaking, in the round, every thread can be regarded as "scheduled" at least once, and to be either executed or found to be disabled.

The key concept of Bounded Unity is the quantified leads-to relation: predicate P *leads to* predicate Q *within* n rounds, notation P **Lt** $\langle n \rangle$ Q . This is defined to mean that every execution fragment that contains a concatenation of n rounds and has its initial state in P, contains a state in Q. The number n is called the *concurrent complexity* of reaching Q from P.

4.2. Bounded unity

The logic of Bounded Unity begins just as UNITY logic with the definitions of a relation **co** and a judgement **transient** for predicates:

$$\begin{array}{lcl} P\ \mathbf{co}\ Q & \equiv & \forall\,(x, y) \in \mathit{step} : x \in P \wedge \mathit{INV} \ \Rightarrow\ y \in Q\ , \\ \mathbf{transient}(P) & \equiv & \exists\ r : P \wedge \mathit{INV} \subseteq \mathit{ena}(r)\ \wedge\ (\forall\,(x, y) \in \mathit{fwd}(r) : x \in P \wedge \mathit{INV} \Rightarrow y \notin P)\ . \end{array}$$

P **co** Q means that every step that starts in P ends in Q. The judgement **transient**(P) expresses that there is a specific thread r that, from every state of P, establishes $\neg P$. For UNITY, the operators **co** and **transient** are introduced by Misra [Mis01] with the same informal meanings. The definitions differ, however, because UNITY does not have *step*, *INV*, *ena*, and *fwd*.

The operators **co** and **transient** are combined in the relations **unless** and **ensures** defined by:

$$\begin{array}{lll} P \textbf{ unless } Q & \equiv & (P \wedge \neg Q) \textbf{ co } (P \vee Q)\,, \\ P \textbf{ ensures } Q & \equiv & (P \textbf{ unless } Q) \ \wedge \ \textbf{transient}(P \wedge \neg Q)\,. \end{array}$$

At this point Bounded Unity deviates more strongly from UNITY by defining the leads-to operator **Lt** with a numerical parameter.

The basic proof rules of Bounded Unity are

- If P **ensures** Q, then P **Lt** $\langle 1\rangle$ Q.
- If $P \wedge \mathit{INV} \subseteq Q$, then P **Lt** $\langle n\rangle$ Q for every $n \geq 0$.
- If P **Lt** $\langle k\rangle$ Q and Q **Lt** $\langle m\rangle$ R, then P **Lt** $\langle k + m\rangle$ R.
- For any family $(P_i)_{i\in I}$, if P_i **Lt** $\langle n\rangle$ Q for all $i \in I$, then $(\exists\ i \in I : P_i)$ **Lt** $\langle n\rangle$ Q.

The first rule is called the **ensures** rule, the second one is the subset rule, the third one is called transitivity, the last one is the disjunction rule.

There is also the Progress-Safety-Progress Rule [CM88]:

PSP: $\quad (P \textbf{ Lt } \langle n\rangle\ Q) \ \wedge \ (A \textbf{ unless } M) \quad \Rightarrow \quad (P \wedge A) \textbf{ Lt } \langle n\rangle\ ((Q \wedge A) \vee M).$

An easy special case is

PSP0: $\quad (\mathit{true} \textbf{ Lt } \langle n\rangle \neg P) \ \wedge \ (A \textbf{ unless } M) \ \wedge \ (A \subseteq P) \quad \Rightarrow \quad (A \textbf{ Lt } \langle n\rangle\ M).$

Here, *true* is the predicate "everywhere true", rather than the boolean value. Predicate P is called *temporary* iff it satisfies *true* **Lt** $\langle n\rangle \neg P$ for some number n.

The soundness of these proof rules for the operational semantics of Sect. 4.1 has been proved mechanically with PVS, see [Hes16b].

Remark. There are several differences between UNITY and Bounded Unity. UNITY is a syntactic formalism, Bounded Unity is a semantic theory founded on operational semantics. In particular, UNITY has its Substitution Axiom [CM88, p. 49] that allows using invariants everywhere in the formalism, whereas Bounded Unity explicitly allows the use of INV at certain points. Bounded Unity is more general than UNITY in that it allows non-forward steps. UNITY makes its statements total, whereas Bounded Unity allows commands to be disabled; this point is more a matter of convenience than a principal difference. The main difference, however, is that Bounded Unity gives the leads-to operator a numerical parameter. □

The next lemma is about the growth of a numerical state function. More specifically, it says that, if the state space can be divided in parts, and a state function grows on each part in a prescribed way, then it grows overall in some way. The reader may skip this result, and return to it later when needed.

Lemma 1 Let vf be an integer-valued state function. Let $(Q_i)_{i\in I}$ be a family of predicates, with $0 \in I$. Let $D \in \mathbb{N}$. For each $i \neq 0$ in I, let $d_i \in \mathbb{N}$ be such that $0 < d_i \leq D$.

Assume that the family covers the state space in the sense that $\mathit{INV} \subseteq (\exists\ i \in I : Q_i)$. Assume, for all $k \in \mathbb{Z}$ and $i \neq 0$, that

$$(3) \qquad ((k \leq \mathit{vf}) \ \wedge \ Q_i) \quad \textbf{Lt } \langle d_i\rangle \quad (k + d_i \leq \mathit{vf})\,.$$

Then, for all $n \geq 0$ and $k \in \mathbb{Z}$, it holds that

$$(4) \qquad (k \leq \mathit{vf}) \quad \textbf{Lt } \langle n + D - 1\rangle \quad ((k + n \leq \mathit{vf}) \ \vee \ Q_0)\,.$$

Assumption (3) prescribes the growth of vf in a distributed way with delay numbers d_i. The intuition behind the lemma is that Formula (3) is applied repeatedly with transitivity. The number of rounds $n + D - 1$ is needed to allow for overshooting. Q_0 is an alternative without progress.

To get some of the intuition, consider the problem of building a wall of depth n with sufficiently many building blocks of differents depths $d_i \leq D$. Depth n is reachable, but the result may be a ragged wall with different depths at different places. The wall need never be higher than $n + D - 1$. The proof is by induction, first covering the ground with single blocks, and then using the induction hypothesis.

Proof The proof is by induction over n. Assume that formula (4) holds for all natural numbers $n < n'$ and all k. It suffices to prove that formula (4) holds for $n := n'$ and a given number k'. If $n' = 0$, the assertion follows from the subset rule. Therefore, assume that $n' > 0$.

For all $i \in I$, we define $P_i = ((k' \leq vf) \wedge Q_i)$. As the family $(Q_i)_{i \in I}$ covers the state space, we have that $(\exists\ i : P_i) = (k' \leq vf)$. By the disjunction rule, it now suffices to prove, for all i, that

$$P_i \quad \mathbf{Lt}\,\langle n' + D - 1\rangle \quad ((k' + n' \leq vf) \ \vee \ Q_0)\,. \tag{5}$$

For $i = 0$, Formula (5) follows from the subset rule. It remains to consider $i \neq 0$. Assumption (3) now implies that

$$P_i \quad \mathbf{Lt}\,\langle d_i\rangle \quad (k' + d_i \leq vf)\,. \tag{6}$$

If $n' \leq d_i$, this implies Formula (5) by means of the subset rule together with transitivity, because $d_i \leq n' + D - 1$ and $k' + n' \leq k' + d_i$. Otherwise, we have $d_i < n'$, and the induction hypothesis (4) with $n := n' - d_i < n'$ and $k := k' + d_i$ gives

$$(k' + d_i \leq vf) \quad \mathbf{Lt}\,\langle n' - d_i + D - 1\rangle \quad ((k' + n' \leq vf) \ \vee \ Q_0)\,. \tag{7}$$

The Formulas (6) and (7) imply Formula (5) by transitivity.

4.3. Progress of commands

In Fig. 1, the abstract commands are not forward steps in the sense of Sect. 4.1, but they are supposed to be implemented by such steps in an unspecified way. Therefore, parameters are introduced to specify their concurrent complexity.

We assume that being at the critical section is temporary, i.e., that the critical section terminates within a given number *cs* of rounds:

$$\mathit{true} \quad \mathbf{Lt}\,\langle \mathit{cs}\rangle \quad \neg(q\ \mathbf{at}\ 16)\,. \tag{8}$$

Here, the antecedent *true* is preferred instead of q **at** 16, but both versions would express the same thing.

As the mutual exclusion algorithm *MXd* used at the nodes is unknown, we introduce parameters W and E for the concurrent complexity of the waiting section and the exit protocol of *MXd*, respectively. If we use the labels in Fig. 1, this amounts for the exit protocol to the assumption

$$\mathit{true} \quad \mathbf{Lt}\,\langle E\rangle \quad \neg(q\ \mathbf{at}\ 18)\,. \tag{9}$$

The waiting section is more complicated. We can only assume that, when the node is free and thread q is waiting, the node will be occupied by some thread within W rounds. Moreover, if thread q is waiting at 15, it is waiting at node $n = \mathit{path}(q, \ell.q)$, but step 15 of q modifies $\ell.q$, making $n \neq \mathit{path}(q, \ell.q)$. We therefore introduce a free variable k for the value of $\ell.q$, and express the progress assumption at line 15 by

$$(q\ \mathbf{at}\ 15 \ \wedge \ \ell.q = k) \quad \mathbf{Lt}\,\langle W\rangle \quad (\mathtt{mu}[\mathit{path}(q, k)] \neq \bot)\,. \tag{10}$$

Note that this does not say that thread q does a step: in the postcondition q may still be waiting at 15, in which case some other thread occupies $\mathit{path}(q, k)$.

The transition system of Fig. 1 also has commands at lines 12, 17, and 19. These commands are regarded as forward commands in the sense of Sect. 4.1. The ensures rule is used to prove that being at 12, 17, or 19 is temporary, i.e., for each $k \in \{12, 17, 19\}$,

$$\mathit{true} \quad \mathbf{Lt}\,\langle 1\rangle \quad \neg(q\ \mathbf{at}\ k)\,. \tag{11}$$

4.4. Parameters for Peterson's algorithm

In this section, we determine values for the parameters E and W for the case that *MXd* is Peterson's algorithm as given in Sect. 2.2. It suffices to consider an abstract setting with just one incarnation of the algorithm. In other words, we can ignore the node, and use the shared variables

$\mathtt{turn} : \mathit{Child} := 0$;
$\mathtt{flag}[\mathit{Child}] : \mathit{bool} := \mathit{false}$;
$\mathtt{mu} : \mathit{Child} \cup \{\bot\} := \bot$.

Here `mu` is the history variable of Sect. 2.3, needed to express property (10).

The two-thread algorithm has the transition system

$$
\begin{array}{ll}
& \mathit{AlgPeterson}(i : \mathit{Child}) = \\
21 & \quad \mathit{NCS} \,; \\
22 & \quad \mathtt{flag}[i] := \mathit{true} \,; \\
23 & \quad \mathtt{turn} := 1 - i \,; \\
24 & \quad \textbf{await } (\neg\, \mathtt{flag}[1-i] \ \vee\ \mathtt{turn} = i) \textbf{ then } \mathtt{mu} := i \,; \\
25 & \quad \mathit{CS} \,; \\
26 & \quad \mathtt{mu} := \bot \,;\ \mathtt{flag}[i] := \mathit{false} \,;\ \textbf{goto } 21 \,.
\end{array}
$$

Line 24 violates the principle of single critical reference (see Sect. 1): it reads two shared variables $\mathtt{flag}[1-i]$ and $\mathtt{turn}$. This violation can be allowed because it concerns a disjunction: the step can be taken when either of the tests succeeds. The assignments to $\mathtt{mu}$ are harmless because $\mathtt{mu}$ is a history variable.

Line 26 corresponds to line 18 of Fig. 1. As it contains a single assignment, thread i traverses line 26 within a single round. This implies that Formula (9) holds for $E = 1$.

For completeness, and as a preparation for the treatment of the waiting section, safety of the algorithm is proved first. Mutual exclusion is implied by the invariant

Jq0: $\quad i \textbf{ in } \{25, 26\} \ \Rightarrow\ \mathtt{mu} = i \,.$

This predicate is threatened only by step 24. It is preserved because of the invariants

Jq1: $\quad \mathtt{flag}[i] \ \equiv\ (i \textbf{ in } \{23 \ldots 26\}) \,,$

Jq2: $\quad i \textbf{ in } \{25, 26\} \ \wedge\ 1 - i \textbf{ at } 24 \ \Rightarrow\ \mathtt{turn} = i \,.$

Predicate *Jq1* is inductive. Predicate *Jq2* is threatened only by step 24. It is preserved because of *Jq1*. This proves the validity of the three invariants *Jq**.

The waiting section, line 15 of Fig. 1, corresponds here to the fragment of the three lines 22, 23, 24. It is clear that the first two lines only require two rounds. We thus have

(12) $\quad (i \textbf{ in } \{22, 23\}) \quad \textbf{Lt } \langle 2 \rangle \quad (i \textbf{ at } 24) \,.$

For line 24, we claim that

(13) $\quad (i \textbf{ at } 24) \quad \textbf{Lt } \langle 3 \rangle \quad (\mathtt{mu} \neq \bot) \,.$

The proof of this formula requires a case distinction. Consider the following predicates

$$
\begin{array}{ll}
P_1(i) : & \quad i \textbf{ at } 24 \ \wedge\ \neg\, \mathtt{flag}[1-i] \,, \\
P_2(i) : & \quad i \textbf{ at } 24 \ \wedge\ 1 - i \textbf{ at } 23 \,, \\
P_3(i) : & \quad i \textbf{ at } 24 \ \wedge\ 1 - i \textbf{ at } 24 \ \wedge\ \mathtt{turn} = i \,, \\
Q : & \quad \mathtt{mu} \neq \bot \,.
\end{array}
$$

At this point, UNITY logic is indispensable. Indeed, it can be proved that

$$
\begin{array}{lll}
P_1(i) & \textbf{ensures} & P_2(i) \vee Q \,, \\
P_2(i) & \textbf{ensures} & P_3(i) \vee Q \,, \\
P_3(i) & \textbf{ensures} & Q \,.
\end{array}
$$

By the inference rules of Bounded Unity, it follows that

$$P_1(i) \vee P_2(i) \vee P_3(i) \quad \textbf{Lt } \langle 3 \rangle \quad Q \,.$$

Using the invariants *Jq0* and *Jq1*, one proves that

$$(i \textbf{ at } 24) \wedge \mathit{INV} \quad \subseteq \quad P_1(i) \vee P_2(i) \vee P_3(i) \vee P_3(1-i) \vee Q \,.$$

Finally, the disjunction rule is used to infer Formula (13).

The Formulas (12) and (13) together imply

(14) $\quad (i \textbf{ in } \{22, 23, 24\}) \quad \textbf{Lt } \langle 5 \rangle \quad (\mathtt{mu} \neq \bot) \,.$

As the lines 22–24 correspond to line 15 of Fig. 1, this proves that Peterson's algorithm for *MXd* satisfies Formula (10) for the value $W = 5$. The results of this section have been proved in a separate PVS proof script, available at [Hes16b].

The parameters for Peterson's algorithm are better than those for Dekker's algorithm. Indeed, for Dekker's algorithm, the Formulas (9) and (10) hold for $E = 2$ and $W = 14$, respectively. The proof is somewhat longer than the proof for Peterson's algorithm.

5. Throughput

The throughput of the algorithm is the number of times threads have returned to the noncritical section. In line 19 of Fig. 1, a private history variable *cnt* counts the number of times thread p returns to the noncritical section. It follows that the throughput is counted by the sum $sinc = \sum_q cnt.q$. In order to compare the throughput with the number of consecutive rounds, we express the growth of *sinc* in terms of the leads-to operator **Lt**.

The predicate $AI = (\forall\ q : q\ \mathbf{at}\ 11)$ expresses that all threads are idle. Of course, we cannot expect progress, i.e., growth of *sinc*, when *AI* holds.

5.1. Counting steps

The function *sinc* only grows in line 19 of Fig. 1, but the other steps must also be counted. For this purpose, a function *savf* is constructed, which grows as fast as the number of rounds (unless all threads are idle), and which is proportional to *sinc* up to a bounded difference. This implies how *sinc* grows.

For the construction of *savf*, we first form a state function $lvf(q)$ that equals 0 when thread q is at line 11, that increases with 1 when thread q does one of the forward steps 12 and 17, that increases with W when it does step 15, with *cs* when it does step 16, and with E when it does step 18, that remains constant when q does step 11, and also when some thread $p \neq q$ does a step. One can verify that these requirements are fulfilled by

$$\begin{aligned} lvf(q) = (\ & pc.q < 16\ ?\quad (\delta(q) - 1 - \ell.q) \cdot (W + 1) + (pc.q = 15\ ?\ 1\ :\ 0) \\ & :\ pc.q = 16\ ?\quad \delta(q) \cdot (W + 1) + 1 \\ & :\ pc.q < 19\ ?\quad cs + \delta(q) \cdot (W + 1) + (E + 1) \cdot (\ell.q + 1) + pc.q - 16 \\ & :\ cs + 2 + \delta(q) \cdot (W + E + 2))\ . \end{aligned}$$

Here, as in the programming language C, an expression $B\ ?\ X\ :\ Y$ is a conditional expression meaning X if B holds, and otherwise Y. Note that Y itself can be a conditional expression.

Let *Depth* be the depth of the tree, i.e., the maximum of the numbers $\delta(q)$ for all leaves q. Then we have

$$0 \leq lvf(q) < A \text{ where } A = cs + 3 + Depth \cdot (W + E + 2). \tag{15}$$

In line 19 of Fig. 1, thread p increments its private variable $cnt.p$. It follows that the function

$$avf(q) = lvf(q) + A \cdot cnt.q$$

increases in every forward step of thread q; it increases with W, *cs*, E in the steps 15, 16, and 18 of thread q, and it remains constant under the steps 11 of q and under all steps of threads $p \neq q$.

It follows that the sum $savf = \sum_q avf(q)$ increases with 1 in every forward step of the algorithm, with W, *cs*, E whenever some thread does a step 15, 16, 18, respectively, and remains constant under steps 11.

Using the assumptions (8), (9) and the PSP0-rule, it follows that

$$\begin{aligned} &(k \leq savf\ \wedge\ q\ \mathbf{at}\ 16)\quad \mathbf{Lt}\,\langle cs\rangle\quad (k + cs \leq savf)\ , \\ &(k \leq savf\ \wedge\ q\ \mathbf{at}\ 18)\quad \mathbf{Lt}\,\langle E\rangle\quad (k + E \leq savf)\ . \end{aligned}$$

In the same way, for each line number $j \in \{12, 17, 19\}$, Formula (11) gives by the PSP0-rule the formula

$$(k \leq savf\ \wedge\ q\ \mathbf{at}\ j)\quad \mathbf{Lt}\,\langle 1\rangle\quad (k + 1 \leq savf)\ .$$

We eliminate the variables q and j from the above formulas by the disjunction rule to obtain

$$\begin{aligned} &(k \leq savf\ \wedge\ CovCS)\quad \mathbf{Lt}\,\langle cs\rangle\quad (k + cs \leq savf)\ , \\ &(k \leq savf\ \wedge\ CovE)\quad \mathbf{Lt}\,\langle E\rangle\quad (k + E \leq savf)\ , \\ &(k \leq savf\ \wedge\ Cov1)\quad \mathbf{Lt}\,\langle 1\rangle\quad (k + 1 \leq savf)\ , \end{aligned} \tag{16}$$

where the predicates *CovCS*, *CovE*, *Cov1* are defined by

$$\begin{aligned} CovCS &= (\exists\ q : q\ \mathbf{at}\ 16)\ , \\ CovE &= (\exists\ q : q\ \mathbf{at}\ 18)\ , \\ Cov1 &= (\exists\ q : q\ \mathbf{in}\ \{12, 17, 19\})\ . \end{aligned}$$

5.2. Waiting for throughput

Of course, the main problem for progress is the waiting at line 15. If $\mathtt{mu}[n] = \bot$ for some node n, this can only be falsified by a step at line 15. Such a step increases *savf* with W. We therefore have

$$(k \le savf \ \wedge \ q \ \textbf{at}\ 15 \ \wedge \ \ell.q = j \ \wedge \ \mathtt{mu}[path(q,j)] = \bot) \quad \textbf{unless} \quad (k + W \le savf)\,.$$

This is combined with assumption (10) and the PSP-rule to give

$$(17) \qquad (k \le savf \ \wedge \ q \ \textbf{at}\ 15 \ \wedge \ \ell.q = j \ \wedge \ \mathtt{mu}[path(q,j)] = \bot) \quad \textbf{Lt}\,\langle W\rangle \quad (k + W \le savf)\,.$$

This formula is used to obtain

$$(18) \qquad (k \le savf \ \wedge \ CovW) \quad \textbf{Lt}\,\langle W\rangle \quad (k + W \le savf)\,, \text{ where}$$
$$CovW = (\forall \ q : q \ \textbf{in}\ \{11, 15\}) \wedge (\exists \ q : q \ \textbf{at}\ 15)\,.$$

Formula (18) is proved from (17) by the disjunction rule. Indeed, it suffices to show that *CovW* together with the invariants implies the existence of a thread q and a number m such that q is at 15 and $\ell.q = m$ and $\mathtt{mu}[path(q,m)] = \bot$. As there are threads at line 15 by *CovW*, we can define m_0 to be the minimum of the numbers $\ell.q$ where q ranges over the threads at line 15. Let q_0 be a thread at line 15 with $\ell.q_0 = m_0$. We have $m_0 \ge 0$ because of *Iq1*. It remains to show that $\mathtt{mu}[path(q_0, m_0)] = \bot$.

Assume that $\mathtt{mu}[path(q_0, m_0)] \neq \bot$. Then *Kq1* implies the existence of a thread r with $\ell.r < m_0 < \delta(r)$. By *Iq4*, thread r is in $\{12 \ldots 18\}$. Predicate *CovW* therefore implies that thread r is at line 15. Then $\ell.r < m_0$ contradicts the minimality of m_0. This proves that $\mathtt{mu}[path(q_0, m_0)] = \bot$, thus completing the proof of (18).

5.3. Estimating throughput

If a state x is not in *CovCS*, *CovE*, *Cov1*, then, in state x, all threads are at 11 or 15 by *Kq0*. Therefore, either all threads are at 11 and $x \in AI$, or $x \in CovW$. This proves that the sets *AI*, *Cov1*, *CovCS*, *CovE*, and *CovW* cover the state space in the sense that *INV* is contained in their union. Therefore, by the Formulas (16) and (18), Lemma 1 applies and gives

$$(19) \qquad (k \le savf) \quad \textbf{Lt}\,\langle n + M - 1\rangle \quad ((k + n \le savf) \ \vee \ AI)\,,$$

where M is the maximum of the three parameters W, cs, and E. This shows that, after an initial delay of at most $M - 1$ rounds, the function *savf* grows as fast as the number of rounds, unless all threads are idle.

We finally need to relate function *savf* to the throughput function *sinc*. Using Formula (15), it is easy to see that

$$(20) \qquad A \cdot sinc \le savf\,.$$

As the number of competing threads can be much smaller than the total number of threads N, we introduce #*comp* for the number of competing threads. Because $lvf(q) = 0$ when q is idle, it follows from Formula (15) that

$$savf \ \le \ A \cdot sinc + (A - 1) \cdot \#comp\,,$$

We introduce a free variable c for comparison with #*comp* and get

$$(21) \qquad (savf \ \le \ A \cdot sinc + (A - 1) \cdot c) \ \vee \ (c < \#comp)\,.$$

In view of the subset rule and the transitivity rule of Bounded Unity, we observe

$$\begin{array}{ll} & (k \le sinc) \\ \subseteq & \{\ \text{Formula (20)}\ \} \\ & (A \cdot k \le savf) \\ \textbf{Lt}\,\langle n + M - 1\rangle & \{\ \text{Formula (19)}\ \} \\ & (A \cdot k + n \le savf) \ \vee \ AI \\ \subseteq & \{\ \text{Formula (21)}\ \} \\ & (A \cdot k + n \le A \cdot sinc + (A - 1) \cdot c) \ \vee \ (c < \#comp) \ \vee \ AI \\ \subseteq & \{\ \text{choose } n := A \cdot i + (A - 1) \cdot c \text{ and divide by } A\ \} \\ & (k + i \le sinc) \ \vee \ (c < \#comp) \ \vee \ AI\,. \end{array}$$

By transitivity and substitution of n, we thus obtain the progress formula, for natural numbers k, i, c:

$$(22)\qquad (k \le sinc)\quad \mathbf{Lt}\,\langle\, A\cdot i + H(c)\,\rangle\quad (k+i \le sinc)\ \vee\ Alt(c)\,,\ \text{where}$$
$$H(c) = (A-1)\cdot c + M - 1\,,$$
$$Alt(c) = (c < \#comp\ \vee\ AI)\,.$$

In words, Formula (22) can be rephrased as the following theorem.

Theorem 2 For the tournament algorithm TMX, the goal of increasing $sinc$ by i (i.e., to do a sequence of steps that contains i steps in which a thread returns to NCS) or reaching a state with more than c competing threads or with no competing threads, has the concurrent complexity $A\cdot i + H(c)$.

The throughput is therefore proportional to the number of rounds with a factor A, given in Formula (15). The factor A is proportional to $Depth$. If one uses a balanced tree, it is logarithmic in the number N of threads. The additive constant $H(c)$ is a kind of initial delay, small when the number of competing threads is small. Note that $Alt(N) = AI$.

6. Complexity of individual progress

For a mutual exclusion algorithm, *individual progress* (also called lockout freedom [Lam86]) means that every competing thread eventually arrives at the critical section and returns to the noncritical section. If the algorithm MXd has individual progress, the tournament algorithm of Fig. 1 also has individual progress. A measure for the individual progress is the individual delay, the smallest number of rounds that guarantees that every thread is idle at least once.

The worst-case individual delay cannot be less than $N\cdot(W+E)$, because when some thread q starts waiting, $N-1$ other threads may have started waiting just before q, and may have to go through the waiting and exit sections before thread q can do this.

In this section, we first sketch an abstract way to compute an upper bound for the individual delay, followed by a full treatment for the case that algorithm MXd satisfies FCFS. In the latter case, if the tree is balanced, the tournament algorithm has a worst-case delay close to optimal.

6.1. Individual delay calculated at the back of an envelope

In this section the individual delay is calculated by a rough and dirty method, with two steps that I cannot formally justify. It may well be that both steps can be justified by rely-guarantee methods as proposed, e.g., by Xu et al. [XdRH97], but this goes beyond the scope of this article.

Assume that the algorithm MXd is such that, for some given constants a and b, when the critical section is always passed within c rounds, then the complete protocol of MXd is passed within $a\cdot c + b$ rounds. This means that MXd satisfies, for every positive number c, the implication

$$(23)\qquad (q\ \mathbf{in}\ CS)\quad \mathbf{Lt}\,\langle c\rangle\quad \neg\,(q\ \mathbf{in}\ CS)$$
$$\Rightarrow\quad (q\ \mathbf{at}\ Entry)\quad \mathbf{Lt}\,\langle\, a\cdot c + b\,\rangle\quad (q\ \mathbf{after}\ Exit)\,.$$

Consider for TMX the assertion

$$\varphi(k,H):\qquad (q\ \mathbf{at}\ 12\ \wedge\ \ell.q = k)\quad \mathbf{Lt}\,\langle H\rangle\quad (q\ \mathbf{at}\ 17\ \wedge\ \ell.q = k)\,.$$

If $k < \delta(q) - 1$, the steps taken in $\varphi(k,H)$ can be regarded as a nested critical section for MXd and implication (23) seems to imply

$$(24)\qquad \varphi(k,H)\ \wedge\ k < \delta(q) - 1\ \Rightarrow\ \varphi(k+1, a\cdot H + b)\,,$$

where the constant b may have to be adapted to include additional administration. This is the step we cannot formally justify. We now define recursively:

$$f(a,b,c,0) = c\,,$$
$$f(a,b,c,k+1) = a\cdot f(a,b,c,k) + b\,.$$

If $a > 1$, it is easy to prove by induction over k the equality

$$f(a, b, c, k) = a^k \cdot \left(c + \tfrac{b}{a-1}\right) - \tfrac{b}{a-1} \ .$$

Using (8), (24) and the recursive definition of f, we obtain

$$\begin{array}{l} (q \textbf{ at } 12 \ \wedge \ \ell.q = \delta(q) - 1) \\ \textbf{Lt}\,\langle\, f(a, b, cs, \delta(q))\,\rangle \quad (q \textbf{ at } 17 \ \wedge \ \ell.q = \delta(q) - 1) \ . \end{array}$$

If $\delta(q) = 0$, there is only one thread and it needs $cs + 3$ rounds to become idle. This suggests (again without formal proof) that

$$\begin{array}{ll} (25) & \mathit{true} \quad \textbf{Lt}\,\langle\, c(q)\,\rangle \quad (q \textbf{ at } 11)\,, \text{ where} \\ & c(q) = f(a, b, cs, \delta(q)) + 3 = a^{\delta(q)} \cdot \left(cs + \tfrac{b}{a-1}\right) - \tfrac{b}{a-1} + 3 \ . \end{array}$$

If the tree is perfectly balanced, the number of threads (leaves) is $N = d^{\delta(q)}$. It follows that $a^{\delta(q)} = N^e$ where the exponent e is given by $e = \log_d a$. We then have

$$c(q) = N^e \cdot \left(cs + \tfrac{b}{a-1}\right) - \tfrac{b}{a-1} + 3 \ .$$

If we take Dekker's algorithm for *MXd*, then $d = 2$. In [BDH16, Section 4], we found the individual delay to be bounded by $3 \cdot cs + 34$. For the tournament based in Dekker's algorithm, this gives an individual delay of the order of N^e where $e = \log_2 3$.

6.2. Using FCFS

Recall that, for a mutual exclusion algorithm, the first-come first-served property FCFS is defined as follows [Lam74]. It is required that the program fragment *Entry* is a sequential composition of two fragments *Doorway* and *Waiting*, such that *Doorway* is wait-free and that, when a thread has passed *Doorway*, it will enter *CS* before any other thread that is currently not in *Entry*.

For example, Peterson's algorithm given in Sect. 2.2, satisfies FCFS. The Doorway consists of the two assignments to `flag` and `turn`. Indeed, these assignments are wait-free, and, when thread i has done them while the other thread has not yet entered, it will complete *EntryP* before the other thread. The fact that Peterson's algorithm implements FCFS has been proved mechanically in a PVS proof script available at [Hes16b].

Dekker's algorithm does not satisfy FCFS, see [BDH16]. There are several mutual exclusion algorithms with FCFS, e.g. [Lam74, LH91, Tau04].

To model FCFS for *MXd* in a tournament setting, every child that enters the Doorway of a node needs to register the set of the currently waiting siblings, and it can only be allowed to occupy the node when this set has become empty. This can be established by using a shared history variable $\texttt{cur}[n]$ for the set of the currently waiting siblings at node n, and a shared history variable $\texttt{prio}[n, i]$ for the waiting siblings that have priority over child i at node n.

Loop 12, 15 of Fig. 1 is therefore replaced by the loop given in Fig. 2. The treatment of the history variable $\texttt{mu}[n]$ is unchanged. At the end of the Doorway, in line 14, child i adds its identifier to $\texttt{cur}[n]$ because it starts waiting. After waiting, in line 15, it removes its identifier from $\texttt{cur}[n]$, and from all sets $\texttt{prio}[n, j]$ because the siblings j need no longer give priority to child i. At the start of the Doorway, in line 13, child i copies $\texttt{cur}[n]$ to $\texttt{prio}[n, i]$. Child i can stop waiting at line 15 when the critical section is free and $\texttt{prio}[n, i]$ is empty. When $\texttt{mu}[n] = i$ at line 15, the child need not wait because the algorithm *MXd* does not distinguish different threads acting as the same child. Initially, all sets `cur` and `prio` are empty.

If the lines 13 and 14 are combined in a single atomic command, the algorithm would be FIFO (first-in first-out). As we want to model FCFS, these lines must be separate atomic commands. In particular, it must be allowed that different threads arrive at the **await** statement with the same sets `prio`. The last two lines can be separate atomic commands. It is more convenient, however, to include them in the command of the **await** statement; this is allowed because `cur` and `prio` are history variables. There is no need to modify command *Exit*.

Apart from additional administration, the only change in the algorithm is that step 15 is disabled when $\texttt{prio}[n, i]$ is nonempty. Therefore, for safety, nothing changes.

The result for throughput would remain valid if the location 15 is replaced by 13 in the progress assumption (10).

```
12        while ℓ ≥ 0 do
13            n := path(p, ℓ) ; i := sib(p, ℓ) ; prio[n, i] := cur[n] ;
14            add i to cur[n] ;
15            await ((mu[n] = ⊥ ∧ prio[n, i] = ∅) ∨ mu[n] = i) then
                  mu[n] := i ; remove i from cur[n] ;
                  for each j do remove i from prio[n, j] end ;
                  ℓ := ℓ − 1 ;
          endwhile .
```

Fig. 2. The entry protocol with FCFS

For the calculation of individual progress, the progress parameters need to be reconsidered. Firstly, the Doorway is now split off from the waiting section. As it is wait-free, we may assume that, for some constant D, every thread traverses the Doorway always within D rounds:

(26) $\quad true \quad \mathbf{Lt}\,\langle D\rangle \quad \neg\,(q\ \mathbf{in}\ \{13, 14\})$.

On the other hand, Formula (10) now applies the remainder of the waiting section, i.e., to line 15 of Fig. 2.

In the case of Peterson's algorithm as described in Sect. 4.4, we can take $D = 2$ and $W = 3$, because of the Formulas (12) and (13), respectively. Indeed, the Doorway consists of the two lines 22, 23, and line 15 of Fig. 2 corresponds to line 24 of Sect. 4.4.

For the proof of individual progress, we need two more invariants

Lq0: $\quad q\ \mathbf{in}\ \{14, 15\} \ \Rightarrow\ sib(q, \ell.q) \notin \mathtt{prio}[path(q, \ell.q), sib(q, \ell.q)]$,
Lq1: $\quad q\ \mathbf{at}\ 15 \ \Rightarrow\ sib(q, \ell.q) \in \mathtt{cur}[path(q, \ell.q)]$.

Roughly speaking, *Lq0* expresses that a child need never give priority to itself. *Lq1* expresses that $\mathtt{cur}[n]$ contains all waiting children of node n.

The correctness heavily relies on the observation that there is never more than one thread in a single child, as expressed by the predicate

MX2: $\quad q\ \mathbf{in}\ \{13 \ldots 15\} \ \wedge\ r\ \mathbf{in}\ \{13 \ldots 15\} \ \wedge\ path(q, \ell.q) = path(r, \ell.r) \ \wedge\ sib(q, \ell.q) = sib(r, \ell.r)$
$\Rightarrow \quad q = r$.

Predicate *MX2* follows from Formula (1) and the invariants *Iq0*, *Iq1*, *MX1*.

Predicate *Lq0* is threatened only by step 13. It is preserved because of *MX2* and the new invariant

Lq2: $\quad i \in \mathtt{cur}[n] \ \Rightarrow\ \exists\ q : q\ \mathbf{at}\ 15 \ \wedge\ n = path(q, \ell.q) \ \wedge\ i = sib(q, \ell.q)$.

This predicate says that all elements of $\mathtt{cur}[n]$ are waiting children.

Predicates *Lq1* is preserved by step 15 (of other threads) because of *MX2*. Predicate *Lq2* is inductive.

6.3. Calculation of individual progress

In this section, we calculate the concurrent complexity in the tournament algorithm for any thread to become idle again. The argument uses recursion with a somewhat unexpected induction hypothesis.

When the waiting loop is passed, the thread can jump to the critical section, and execute the exit protocol. The **ensures** rule implies that

$$(q\ \mathbf{at}\ 12 \ \wedge\ \ell.q < 0) \quad \mathbf{Lt}\,\langle 1\rangle \quad (q\ \mathbf{at}\ 16 \ \wedge\ \ell.q = -1)\ ,$$
$$(q\ \mathbf{at}\ 17 \ \wedge\ \ell.q = k < \delta(q) - 1) \quad \mathbf{Lt}\,\langle 1\rangle \quad (q\ \mathbf{at}\ 18 \ \wedge\ \ell.q = k)\ .$$

For steps 16 and 18, we use assumptions (8), (9), and the PSP0 rule to obtain

$$(q\ \mathbf{at}\ 16 \ \wedge\ \ell.q < 0) \quad \mathbf{Lt}\,\langle cs\rangle \quad (q\ \mathbf{at}\ 17 \ \wedge\ \ell.q < 0)\ ,$$
$$(q\ \mathbf{at}\ 18 \ \wedge\ \ell.q = k) \quad \mathbf{Lt}\,\langle E\rangle \quad (q\ \mathbf{at}\ 17 \ \wedge\ \ell.q = k + 1)\ .$$

By transitivity, it follows that

(27) $\quad (q\ \mathbf{at}\ 17 \ \wedge\ \ell.q = k < \delta(q) - 1) \quad \mathbf{Lt}\,\langle E + 1\rangle \quad (q\ \mathbf{at}\ 17 \ \wedge\ \ell.q = k + 1)$.

Using these formulas, transitivity, and the disjunction rule, it is easy to see that

$$(28) \qquad \delta(q) > 0 \;\Rightarrow\; (q \textbf{ in } \{12 \ldots 18\} \;\wedge\; \ell.q < 0) \quad \textbf{Lt}\,\langle E + cs + 2\rangle \quad (q \textbf{ at } 17 \;\wedge\; \ell.q = 0)\,.$$

At this point, we begin with a recursive approach with the induction hypothesis that k and H are natural numbers with

$$(29) \qquad \forall\; q : k < \delta(q) \;\Rightarrow\; (q \textbf{ in } \{12 \ldots 18\} \;\wedge\; \ell.q < k) \quad \textbf{Lt}\,\langle H\rangle \quad (q \textbf{ at } 17 \;\wedge\; \ell.q = k)\,.$$

Indeed, by (28), Formula (29) holds for $k := 0$ and $H := E + cs + 2$.

Now the aim is to show that the induction hypothesis (29) with fixed k and H implies Formula (29) with the substitutions $k := k + 1$ and $H := H'$ for some expression H'.

In order to prove progress at line 15, we first prove that

$$(\ell.q < k < \delta(q) \;\wedge\; n = \mathit{path}(q, k)) \;\textbf{unless}\; (\mathtt{mu}[n] = \bot)\,.$$

This **unless** formula is combined with the induction hypothesis (29) via the PSP-rule and *Iq4* to obtain

$$(\ell.q < k < \delta(q) \;\wedge\; n = \mathit{path}(q, k)) \quad \textbf{Lt}\,\langle H\rangle \quad (\mathtt{mu}(n) = \bot)\,.$$

By the disjunction rule, the subset rule, and the invariants *Kq1* and *Iq4*, it follows that

$$(30) \qquad (\delta(n) = k) \quad \textbf{Lt}\,\langle H\rangle \quad (\mathtt{mu}[n] = \bot)\,.$$

For thread q waiting at line 15 to make progress, we need the condition that there are no competing threads at the same node that need not give priority to q. We therefore introduce the set of these competing threads

$$\begin{aligned}\mathit{cprio}(q) = \{r \mid r \textbf{ in } \{14, 15\} \;\wedge\; \mathit{path}(r, \ell.r) = \mathit{path}(q, \ell.q)\\ \wedge\; \mathit{sib}(q, \ell.q) \notin \mathtt{prio}[\mathit{path}(r, \ell.r), \mathit{sib}(r, \ell.r)]\}\,,\end{aligned}$$

and write $\#\mathit{cprio}(q)$ for the number of elements of this set.

Lemma 3 Assume that q is at line 15.

(a) $1 \le \#\mathit{cprio}(q)$.
(b) $\#\mathit{cprio}(q) \le d$.
(c) If $\#\mathit{cprio}(q) = d$ then $\mathtt{mu}[\mathit{path}(q, \ell.q)] = \bot$.

Proof

(a) As q is at 15, we have $q \in \mathit{cprio}(q)$ by *Lq0*, and hence $\#\mathit{cprio}(q) \ge 1$.
(b) For any node n, let $\mathit{Fan}(n)$ be the set of threads r with $\ell.r \le \delta(n) < \delta(r)$ and $\mathit{path}(r, \delta(n)) = n$. Let f be the function from $\mathit{Fan}(n)$ to the set *Child* given by $f(r) = \mathit{sib}(r, \delta(n))$. This function is injective because of *MX1*. This implies that $\#\mathit{Fan}(n) \le d$. On the other hand, as q is at 15, we have $\mathit{cprio}(q) \subseteq \mathit{Fan}(\mathit{path}(q, \ell.q))$. Therefore $\#\mathit{cprio}(q) \le d$.
(c) If $\mathtt{mu}[\mathit{path}(q, \ell.q)] \neq \bot$, the invariant *Kq1* gives a thread $r \in \mathit{Fan}(\mathit{path}(q, \ell.q))$ with $r \notin \mathit{cprio}(q)$, and hence $\#\mathit{cprio}(q) < d$. □

It follows from *Lq1* that, when q is at 15, the set $\mathit{cprio}(q)$ does not receive new elements. Also, when q is at 15, any thread that sets $\mathtt{mu}[\mathit{path}(q, \ell.q)] \neq \bot$, belongs to $\mathit{cprio}(q)$ and leaves $\mathit{cprio}(q)$. Using the PSP rule and Formula (10), we obtain

$$\begin{aligned}(31) \qquad &(q \textbf{ at } 15 \;\wedge\; \ell.q = k \;\wedge\; \mathtt{mu}[\mathit{path}(q, k)] = \bot \;\wedge\; \#\mathit{cprio}(q) = j)\\ &\textbf{Lt}\,\langle W\rangle \quad (q \textbf{ at } 15 \;\wedge\; \ell.q = k \;\wedge\; \#\mathit{cprio}(q) = j - 1) \;\vee\; (q \textbf{ at } 12 \;\wedge\; \ell.q = k - 1)\,.\end{aligned}$$

Combining Formula (31) with Formula (30) by transitivity, we obtain

$$\begin{aligned}&(q \textbf{ at } 15 \;\wedge\; \ell.q = k \;\wedge\; \#\mathit{cprio}(q) = j)\\ &\textbf{Lt}\,\langle\, H + W\,\rangle \quad (q \textbf{ at } 15 \;\wedge\; \ell.q = k \;\wedge\; \#\mathit{cprio}(q) = j - 1) \;\vee\; (q \textbf{ at } 12 \;\wedge\; \ell.q = k - 1)\,.\end{aligned}$$

By induction over j, with Lemma 3(a) for the base case, this gives

$$(32) \qquad (q \textbf{ at } 15 \;\wedge\; \ell.q = k \;\wedge\; \#\mathit{cprio}(q) = j) \quad \textbf{Lt}\,\langle\, j \cdot (H + W)\,\rangle \quad (q \textbf{ at } 12 \;\wedge\; \ell.q = k - 1)\,.$$

By (31) and Lemma 3(b, c), it follows that

$$(q \textbf{ at } 15 \;\wedge\; \ell.q = k) \quad \textbf{Lt}\,\langle\, (d - 1) \cdot (H + W) + W\,\rangle \quad (q \textbf{ at } 12 \;\wedge\; \ell.q = k - 1)\,,$$

Using assumption (26) and transitivity, we get

$$(q \textbf{ in } \{12 \ldots 15\} \ \wedge \ \ell.q = k) \quad \textbf{Lt} \langle\, 1 + D + (d-1) \cdot (H + W) + W \,\rangle \quad (q \textbf{ at } 12 \ \wedge \ \ell.q = k-1) \,.$$

This is combined with the induction hypothesis (29) by transitivity to give

$$(\ell.q = k \ \wedge \ q \textbf{ in } \{12 \ldots 15\}) \quad \textbf{Lt} \langle\, d \cdot H + d \cdot W + D + 1 \,\rangle \quad (q \textbf{ at } 17 \ \wedge \ \ell.q = k) \,. \tag{33}$$

Write $D_0 = d \cdot W + D + E + 2$. The Formulas (27), (29), and (33) combine to

$$\begin{aligned} &\forall \ q : k+1 < \delta(q) \quad \Rightarrow \\ &\quad (\ell.q < k+1 \ \wedge \ q \textbf{ in } \{12 \ldots 18\}) \quad \textbf{Lt} \langle\, d \cdot H + D_0 \,\rangle \quad (q \textbf{ at } 17 \ \wedge \ \ell.q = k+1) \,. \end{aligned} \tag{34}$$

Formula (34) is the induction hypothesis (29) with the substitution $k := k+1$ and $H := d \cdot H + D_0$.

This justifies the recursive definition

$$H_0 = E + cs + 2 \quad \text{and} \quad H_{k+1} = d \cdot H_k + D_0 \,,$$

because Formula (28) combined with the implication (29) $\Rightarrow$ (34) proves by induction that

$$\begin{aligned} &k < \delta(q) \quad \Rightarrow \\ &\quad (\ell.q < k \ \wedge \ q \textbf{ in } \{12 \ldots 18\}) \quad \textbf{Lt} \langle H_k \rangle \quad (q \textbf{ at } 17 \ \wedge \ \ell.q = k) \,. \end{aligned} \tag{35}$$

Put $k = \delta(q) - 1$. Then $\ell.q \le k$ by *Iq0*. From (33) and (35), we obtain

$$(q \textbf{ in } \{12 \ldots 18\}) \quad \textbf{Lt} \langle d \cdot (H_k + W) + D + 1 \rangle \quad (q \textbf{ at } 17 \ \wedge \ \ell.q = k) \,.$$

Two final steps are needed for q to become idle (if it is not yet idle). This implies

$$\mathit{true} \quad \textbf{Lt} \langle d \cdot (H_k + W) + D + 3 \rangle \quad (q \textbf{ at } 11) \,.$$

The recursive definition easily leads to

$$H_k = d^k \cdot (E + cs + 2 + \tfrac{1}{d-1} \cdot D_0) - \tfrac{1}{d-1} \cdot D_0 \,.$$

In this way, we finally obtain the total concurrent complexity

$$\begin{aligned} &\mathit{true} \quad \textbf{Lt} \langle\, c(q) \,\rangle \quad (q \textbf{ at } 11) \ \text{ where} \\ &c(q) = d^{\delta(q)} \cdot (cs + C_0) - C_0 + 3 \ \text{ and} \\ &C_0 = \tfrac{1}{d-1} \cdot D + \tfrac{d}{d-1} \cdot (W + E + 2) \,. \end{aligned} \tag{36}$$

Formula (36) confirms Formula (25) for the case that *MXd* satisfies FCFS, in which case $a = d$ and $b = D + \delta(q) \cdot (W + E + 2)$.

In words, Formula (36) says

Theorem 4 If *MXd* satisfies FCFS, the tournament algorithm *TMX* has the concurrent complexity $c(q)$ given in Formula (36) for any thread q to become idle again.

In other words, any execution fragment of *TMX* that contains a concatenation of $c(q)$ rounds has at least one state in which thread q is idle.

The most important factor of $c(q)$ is the power $d^{\delta(q)}$. If the tree T is perfectly balanced, then $d^{\delta(q)} = N$, the number of threads. The constant C_0 is independent of the depth of the tree. It is reasonably close to the optimal value $W + E$. It only depends on the parameters d, D, W, and E of algorithm *MXd*. These can be assumed to be small.

7. Conclusion

The throughput factor of Theorem 2 is $A = cs + 3 + \mathit{Depth} \cdot (D + W + E + 2)$. The important factor is *Depth*, the depth of the tree. If the tree is balanced, *Depth* is logarithmic in N, the number of threads.

If algorithm *MXd* satisfies FCFS, Theorem 4 gives the individual delay of at most $c(q) = d^{\delta(q)} \cdot (cs + C_0) - C_0 + 3$ where $C_0 = \tfrac{1}{d-1} \cdot D + \tfrac{d}{d-1} \cdot (W + E + 2)$. If the tree is balanced, this is proportional to N and close to the optimal value $N \cdot (W + E)$. FCFS holds for Peterson's algorithm [Pet81], but not for Dekker's algorithm, see [BDH16]. Indeed, it is likely that the worst-case individual delay for tournaments built on Dekker's algorithm is of the order N^e where the exponent e equals $\log_2 3$.

These numbers can be compared with the numbers obtained for other mutual exclusion algorithms. The Bakery algorithms of Lamport [Lam74] and Taubenfeld [Tau04] have throughput factors linear in N, and individual delay quadratic in N, see [Hes16a]. The algorithm of Lycklama-Hadzilacos [LH91] and all its variants have throughput factors quadratic in N, and individual delay cubic in N, see [Hes15]. All these algorithms are FCFS.

The proof assistant PVS [OSRSC01] was indispensable for us to verify almost all technical results of the article.

The rough and dirty calculation in Sect. 6.1 seems to have merit, but it badly needs a formal and mechanical justification. A more compositional formalism is also needed to formally justify the application of a specific algorithm *MXd* in the tournament.

Note that tournament algorithms never have bounded overtaking. Recall that a mutual exclusion algorithm has bounded overtaking if there is a number B such that, when a thread q has passed the doorway and has started waiting, not more than B other threads can enter *CS* before thread q enters *CS*. The point is that the doorway is the largest initial part of the entry protocol that thread q can pass without waiting. For a tournament algorithm, this is the doorway at the first internal node of its root path. The thread can remain at this node, while threads that need not pass this node, can pass to *CS* unboundedly often. As FCFS implies bounded overtaking, this also implies that tournament algorithms are never FCFS.

Experimental confirmation of high throughput for tournament algorithms is obtained by Buhr et al. [BDH15]. We are not aware of experiments to measure the maximal individual delay. Indeed, it is difficult to experimentally measure the number of rounds without disturbing concurrent computation.

References

[AdBO09] Apt KR, de Boer FS, Olderog E-R (2009) Verification of sequential and concurrent programs. Springer, New York

[BDH15] Buhr PA, Dice D, Hesselink WH (2015) High-performance N-thread software solutions for mutual exclusion. Concurr Comput Pract Exper. 27:651–701. doi:10.1002/cpe.3263

[BDH16] Buhr PA, Dice D, Hesselink WH (2016) Dekker's mutual exclusion algorithm made RW-safe. Concurr Comput Pract Exp 28:144–165. doi:10.1002/cpe.3659

[Bur81] Burns JE (1981) Complexity of communication among asynchronous parallel processes, Ph.D. thesis, School of Information and Computer Science, Georgia Institute of Technology

[CM88] Chandy KM, Misra J (1988) Parallel program design, a foundation. Addison-Wesley, USA

[dB67] de Bruijn NG (1967) Additional comments on a problem in concurrent programming control. Commun ACM 10:137–138 **(Letter to the Editor)**

[Dij65] Dijkstra EW (1965) Solution of a problem in concurrent programming control. Commun ACM 8:569

[Hes99] Hesselink WH (1999) Progress under bounded fairness. Distrib Comput 12:197–207

[Hes15] Hesselink WH (2015) Mutual exclusion by four shared bits with not more than quadratic complexity. Sci Comput Program 102:57–75. doi:10.1016/j.scico.2015.01.001

[Hes16a] Hesselink WH (2016) Correctness and concurrent complexity of the Black-White Bakery algorithm. Formal Aspects Comput 28:325–341. doi:10.1007/s00165-016-0364-4

[Hes16b] Hesselink WH (2016) PVS proof script for: tournaments for mutual exclusion. http://wimhesselink.nl/mechver/tournaments (Accessed 8 Aug 2016)

[Kes82] Kessels DE (1982) Arbitration without common modifiable variables. Acta Inf 17:135–141

[Knu66] Knuth DE (1966) Additional comments on a problem in concurrent programming control. Commun ACM 9:321–322 **(Letter to the Editor)**

[Lam74] Lamport L (1974) A new solution of Dijkstra's concurrent programming problem. Commun ACM 17:453–455

[Lam86] Lamport L (1986) The mutual exclusion problem—part I: a theory of interprocess communication, part II: statement and solutions. J ACM 33:313–348

[LH91] Lycklama EA, Hadzilacos V (1991) A first-come-first-served mutual-exclusion algorithm with small communication variables. ACM Trans Program Lang Syst 13:558–576

[MCS91] Mellor-Crummey JM, Scott ML (1991) Algorithm for scalable synchronization on shared-memory multiprocessors. ACM Trans Comput Syst 9:21–65

[Mis01] Misra J (2001) A discipline of multiprogramming: programming theory for distributed applications. Springer, New York

[OG76] Owicki S, Gries D (1976) An axiomatic proof technique for parallel programs. Acta Inf 6:319–340

[OSRSC01] Owre S, Shankar N, Rushby JM, Stringer-Calvert DWJ (2001) PVS Version 2.4, System Guide, Prover Guide, PVS Language Reference. http://pvs.csl.sri.com

[Pet81] Peterson GL (1981) Myths about the mutual exclusion problem. Inf Process Lett 12:115–116
[PF77] Peterson GL, Fischer MJ (1977) Economical solutions for the critical section problem in a distributed system (extended abstract). In: STOC '77: Proceedings of the 9th annual ACM symposium on theory of computing, New York. ACM, pp 91–97
[Tau04] Taubenfeld G (2004) The Black-White Bakery Algorithm and related bounded-space, adaptive, local-spinning and FIFO algorithms. In: Proceedings of the DISC, LNCS, vol 3274, pp 56–70
[XdRH97] Xu Q, de Roever W-P, He J (1997) The rely-guarantee method for verifying shared variable concurrent programs. Formal Aspects Comput 9:149–174
[ZYC96] Zhang X, Yan Y, Castenada R (1996) Evaluating and designing software mutual exclusion algorithms on shared memory multiprocessors. IEEE Parallel Distrib Technol Syst Appl 4:25–42

Received 18 May 2016
Accepted in revised form 23 November 2016 by Xinyu Feng
Published online 27 January 2017